Federal University of Minas Gerais - UFMG

ProVANT Simulator User Guide

Author: Arthur Viana Lara
Translated by: Daniel Leite Ribeiro
Version: 1.0

March 31, 2018

Abstract

The purpose of this guide is to describe the required procedures for using ProVANT Simulator. Its covers from the installation process to performing tests on control strategies via simulation. The text is organized as follows:

1. The first section presents the context in which this simulation environment is inserted, and also a brief introduction on unmanned aerial vehicles (UAVs);
2. The second section presents the steps required for the installation of ProVANT's simulatioin environment;
3. The third section describes the simulation environment's workflow, detailing each feature of the GUI, such as choosing scenario, model, control strategy and airship.
4. The fourth chapter is the most important for the user. It describes the procedures for implementing new control strategies in the simulation environment;
5. Appendix A explains the file structure behind the UAV models used. In this section is presented the main information about kinematic modelling and about importing files from CAD software into the platform Gazebo;
6. Appendix B describes the plugins used in the simulation environment;
7. Appendices C end D describe the files `CMakeLists.txt` and `package.xml`, respectively.

Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

Unmanned aerial vehicles are airships equipped with embedded systems, sensors and actuators that allow the performance of autonomous or remote controlled flights. They are commonly classified in two groups: rotary-wing vehicles, such as helicopters and quadcopters, and fixed-wing vehicles, such as airplanes.

There is a wide range of applications for UAVs, such as:

- Crop dusting;
- Herd conduction;
- Road monitoring;
- Power lines inspection;
- Supply deliver in difficult access locations.

The simulator presented in this guide is associeted to ProVANT[1]. ProVANT consists of a partnership between Federal University of Santa Catarina (UFSC) and Federal University of Minas Gearis (UFMG), aiming towards research and development of new technologies to improve the performance of UAVs. In this context, ProVANT currently focuses in the development of tilt-rotor UAVs. The tilt-rotor is an airship with hybrid configuration, featuring the main advantages of both fixed-wing and rotary-wing airships, such as lower power consuption during cruise flights and vertical take-off and landing (VTOL). It can operate both indoors and outdoors.

Currently, ProVANT has three design branches:

1. Mechanics/Aerodynamics;

2. Instrumentation/Electronics;

3. Control strategy design and state estimation.

The Mechanic/Aerodynamic design alredy has five UAV versions, which were named ProVANT UAV 1.0, 2.0, 2.1, 3.0 and 4.0, shown in Figures 1.1, 1.2, 1.3, 1.4 and 1.5, respectively. Instrumentation/Electronics is in an advanced stage, having all it's circuits already developed, and improvements are being made on them. As for the Control strategy design and state estimation, several control strategy were developed in the context of this project by master's and doctorate students.

Furthermore, some scientific works were developed. In Donadel (2015), controllers based on the Linear Quadratic Regulator (LQR), linear $\mathcal{H}_\infty$ and linear mixed $\mathcal{H}_2/\mathcal{H}_\infty$ techniques were developed for ProVANT UAV 1.0, aiming towards path tracking. Some of these controllers were validated by experimental flights. In de Almeida Neto (2014), a non-linear control technique based in feedback linearization is presented, dedicated to load transportation in ProVANT UAV 2.0. With the same objective, Alfaro (2016) presentes the development of a controller based in the Model Predictive Control (MPC) technique. In Rego (2016), load transportation in ProVANT UAV 2.0 was addressed, although the path tracking problem was approached from the load's poit of view, to which robust state estimators were developed. In Cardoso (2016), an adaptative control strategy was developed aiming towards path tracking for ProVANT UAV 3.0.

[1]provant.paginas.ufsc.br

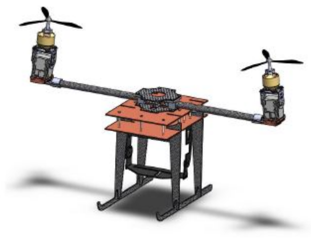


Figure 1.1: ProVANT UAV 1.0's mechanical design.

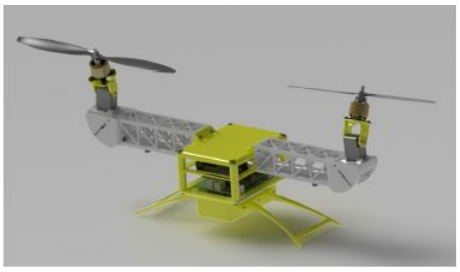


Figure 1.2: ProVANT UAV 2.0's mechanical design..

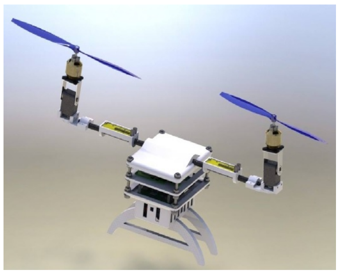


Figure 1.3: ProVANT UAV 2.1's mechanical design..

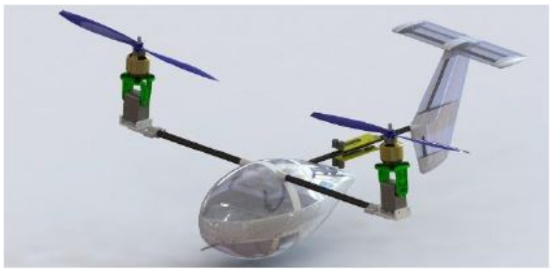


Figure 1.4: ProVANT UAV 3.0's mechanical design.

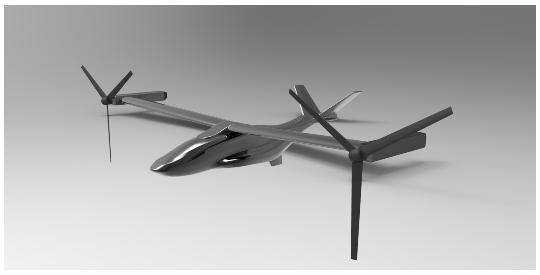


Figure 1.5: ProVANT UAV 4.0's mechanical design.

ProVANT Simulator was made to be a reliable and easy to use tool, allowing for the reduction of costs and time required for the design and validation of control strategies.

Chapter 2

Installation

ProVANT Simulator is a simulation environment developed in order to validate and evaluate the performance of control strategies. The simulator version addressed in this guide was develop on top of Gazebo 7, a simulation platform, and ROS (Robot Operating System), a framework for developing robot applications, under the Kinetic distribution. In order to use it, a computer running the **operating system Ubuntu 16.04** is needed.

ROS offers a programming interface for robotics applications and features repositories with several software modules, and Gazebo is a 3D simulation software under free license, maintained and developed under responsibility of the Open Source Robotics Foundation (OSRF). It's able to simulate the dynamic behavior of rigid, articulated bodies, and includes features such as collision detection and graphical visualization.

This chapter presents the steps required for installation of ProVANT Simulator. It first presents the installation procedures for the platforms used by the simulator, such as Qt5, Git and ROS, and then those for the ProVANT simulation environment.

2.1 Installation procedures

In the steps described here it is assumed that the computer in which the simulator is being installed runs a clean, recently installed copy of Ubuntu 16.04.

2.1.1 Installing Git

She source code for ProVANT simulator is hosted on GitHub. In order to access the source code flies for ProVANT hosted on this GitHub repository, Git must be installed in the user's computer. In case it's not, open a new terminal and run the following commands:

```
sudo apt update
sudo apt install git
```

2.1.2 Installing and configuring ROS

ROS offers a programming interface for robotics applications and several software modules, among which is simulator Gazebo, version 7, used by ProVANT Simulator. This subsection details the instructions needed for installing the ROS Kinetic Kame distribution[1].

Configure the Ubuntu repositories

[1]More details and help can be found on ROS's homepage

Before starting installation, the Ubuntu repositories must be configured to allow *restricted*, *universe* and *multiverse*. Note: Usually, this options are already set upon installation of Ubuntu 16.04. In case they're not, follow the Ubuntu guide for instructions on doing this.

Setup sources.list

The computer must also be set up to accept software from packages.ros.org. To do that, open a new terminal and run the following command:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main"
>/etc/apt/sources.list.d/ros-latest.list'
```

Setup access keys for ROS repositories

Run the following command:

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

Installation

First, make sure the Debian package index is up-to-date:

```
sudo apt update
```

After updating the packages, download the binary files:

```
sudo apt install ros-kinetic-desktop-full
```

Initialize rosdep

Before using ROS, rosdep must be initialized. This system installs the system dependencies for the source code to be compiled. It is required in order to run some core components in ROS.

```
sudo rosdep init
rosdep update
```

Environment setup

To have the ROS environment variables automatically added to the bash session every time a new shell is launched:

```
echo "source /opt/ros/kinetic/setup.bash" >> $HOME/.bashrc
source $HOME/.bashrc
```

Create a ROS workspace

Finally, a ROS workspace must be created. To do that, run the following command sequence:

```
mkdir -p $HOME/catkin_ws/src
cd $HOME/catkin_ws/
catkin_make
source $HOME/catkin_ws/devel/setup.bash
echo "source $HOME/catkin_ws/devel/setup.bash" >> $HOME/.bashrc
```

2.1.3 Installing Qt Framework

In order to appropriately install ProVANT Simulator's graphical setup environment, QtCreator 5 IDE must be installed. To do that, run the following commands:

```
cd $HOME/Downloads
wget http://download.qt.io/official_releases/online_installers/qt-unified-linux-x64-online.run
sudo chmod +x qt-unified-linux-x64-online.run
./qt-unified-linux-x64-online.run
```

2.1.4 Dowloading and installing the simulation environment

The source code for ProVANT Simulator is located in this GitHub repository. Since it's currently private, access must be requested from Professor Guilherme Vianna Raffo[2].

Once it's been granted, the current version of the simulation environment can be downloaded by cloning the repository into the user's computer:

```
cd $HOME/catkin_ws/src
git clone https://github.com/Guiraffo/ProVANT-Simulator.git
```

Once cloned, the simulation environment can be installed and configured by running the following commands:

```
cd $HOME/catkin_ws/src/ProVANT-Simulator
sudo chmod +x install.sh
./install.sh
```

Provided that all the procedures described above were executed successfully, the user will be ready to proceed to the next chapter, regarding ProVANT Simulator's workflow.

[2]raffo@ufmg.br

Chapter 3

Workflow

This chapter presents the features available for ProVANT Simulator's operation through the graphical interface, such as:

- Scenario selection
- UAV model selection
- Scenario configuration
- UAV settings
- Simulation startup

3.1 How to start the simulation environment

Once the installation process has been gone through successfully as described in the previous chapter, the development of controllers in the simulation enviromnent can begin. To start the GUI, open a new terminal and type the following command:

```
provant_gui
```

The graphical interface to access the simulator will then be started, and the main window will show up, as depicted in Figure 3.1.

3.2 Scenario selection

Once the simulator's GUI has been started, the user must select a simulation scenario. That can be done in two different ways, through the option *Simulation*, present in the GUI's top menu, as shown in figure 3.2:

(i) By clicking *New*, a template scenario with extension `.tpl` can be opened.

A template scenario is a scenario configuration that serves as template for the criation of other scenarios. It cannot be edited, needs the inclusion of a UAV model an must be saved before the simulation with format `.world`.

(ii) By clicking *Open*, an existing sncenario (with extension `.world`) can be opened.

Menu *Simulation* also allows to save a scenario with another name under the `.world` extension, by clicking *Save*, and to end the simulation environment, by clicking *Exit*.

3.3 UAV selection

Through option *Edit* in the top menu, the UAV model to be used in the simulation can be selected. Note: in this version of the simulation environment, only a single UAV can be included in a given simulation instance.

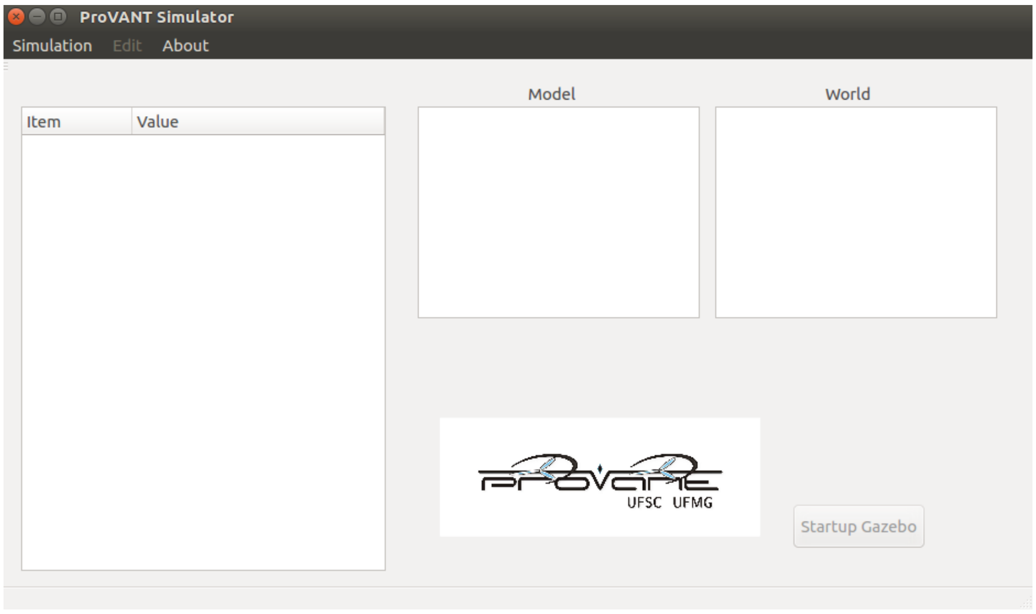


Figure 3.1: Main window

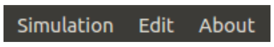


Figure 3.2: Top menu

3.4 Simulation settings window

After choosing the UAV and the simulation scenario, an illustrative image of the scenario will appear in the GUI. Thus, it is possible to verify if it was selected correctly, as shown in Figure 3.3, through item 3.

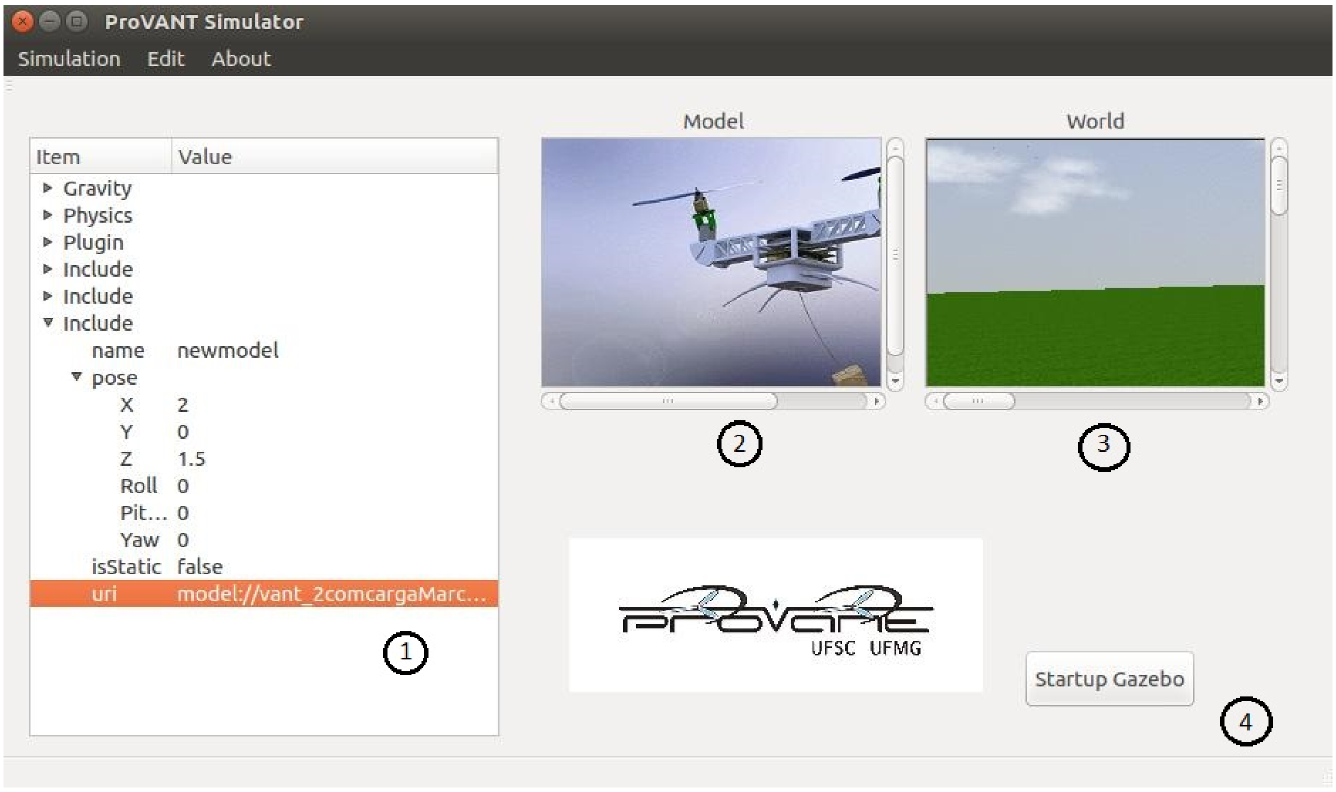
In this same window, item 1 consists of a tree with information and settings for the UAV and the simulation scenario. Settings can be edited by double-clicing the desired field. The main fields of this tree are:

- Gravity: gravity acceleration vector with respect to the world frame, in m/s^2;
- Physics: physics engine used to perform the simulation. The available options are:
  - ode
  - bullet
  - dart
  - simbody
- name: Name of the UAV model in simulator Gazebo
- pose: Initial position and orientation (roll, pitch, yaw) of the UAV, in meters and radians, respectively.

By double-clicking tue *uri* field, a new window will open. In this window it is possible to view and configure the model, controller, actuators and sensors. More details on this window will be presented in the next section. Item 4 allows the initialization of the simulation with the settings, model and scenario selected by the user in the GUI.

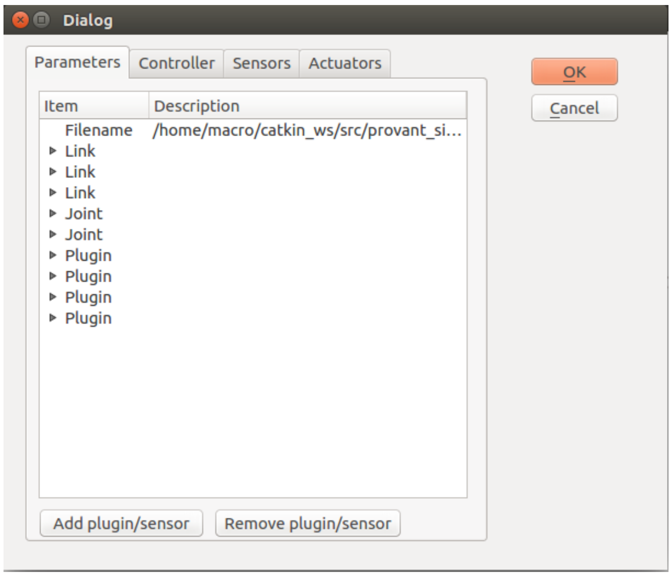
3.5 Model, control strategy and instrumentation visualization and settings window

Figure 3.5 shows tab *Controller*. In this tab, it is possible to create a new contorl strategy, using item 2, select an existent one, through item 3, or compile it, item 4. The control strategi selected for use in the simulation is shown on item 1. More details on these items are fefscribed below.



Item	Value
▸ Gravity	
▸ Physics	
▸ Plugin	
▸ Include	
▸ Include	
▾ Include	
name	newmodel
▾ pose	
X	2
Y	0
Z	1.5
Roll	0
Pit…	0
Yaw	0
isStatic	false
uri	model://vant_2comcargaMarc…

Figure 3.3: Simulation settings window

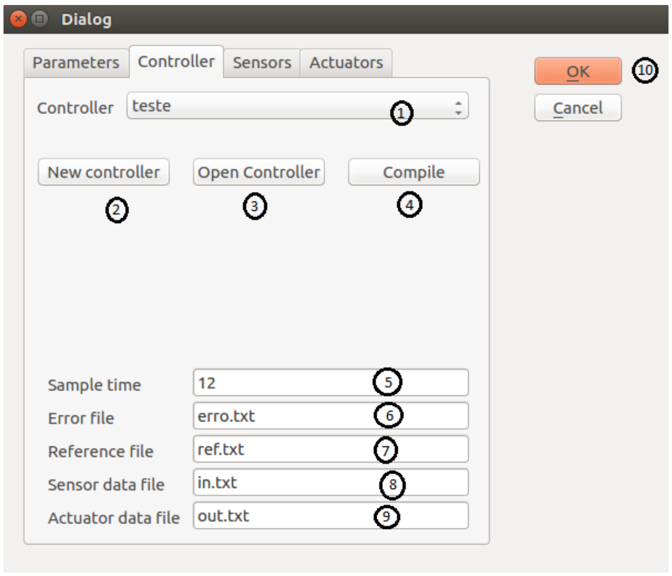


Item	Description
Filename	/home/macro/catkin_ws/src/provant_si…
▸ Link	
▸ Link	
▸ Link	
▸ Joint	
▸ Joint	
▸ Plugin	
▸ Plugin	
▸ Plugin	
▸ Plugin	

Figure 3.4: UAV model parameters visualization tab.

Creating a new control strategy

To create a new control strategy, the button *New controller* (item 2) must be clicked, and a new window will be shown (Figure 3.6). In this window, the user must insert the new control strategy's name (more details on the creation of a new control strategy are presented in Section 4).



Controller: teste

Sample time: 12

Error file: erro.txt

Reference file: ref.txt

Sensor data file: in.txt

Actuator data file: out.txt

Figure 3.5: Control strategy selcection and configuration window.

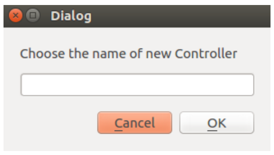


Figure 3.6: Creating a new control strategy.

Modifying an existing control strategy

To modify an ecisting control staregy, the user must select its name among several ones listed in the listing box (item 1) and click the button *Open controller* (item 3). After choosing the strategy, the file manager Nautilus will be opened in the directory containing all the files and directories associated with the controller, as showin in Figure 3.7.

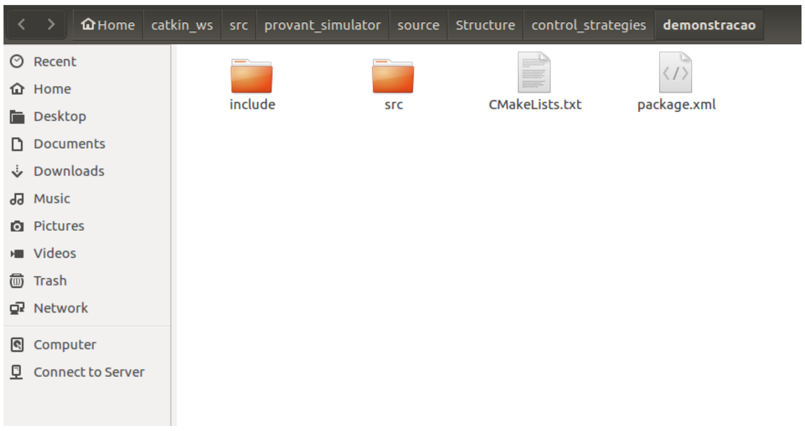


Figure 3.7: Directory containing all the files and folders associated to the control strategy to be modified.

Compiling the contorller

To compile the code associeted to the selected control strategy, the user must click the button *Compile* (item 4). **This step must be executed whenever new modifications to the control strategy are made**. In

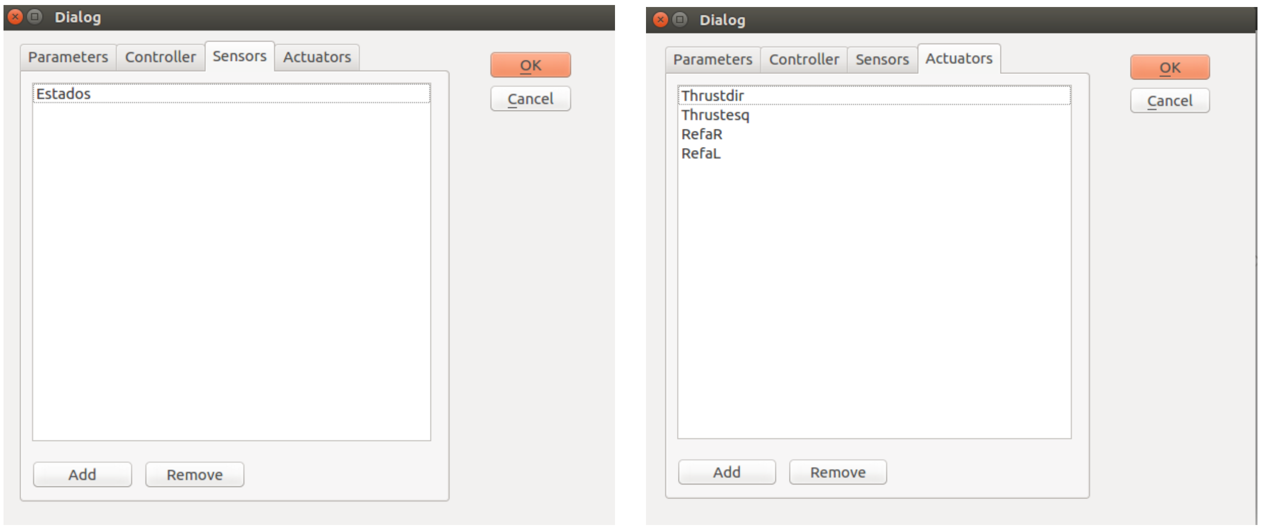


Figure 3.8: Tabs for selecting the available instrumentation during simulation.

case there are errors during compilation, the compiler's output log wil be shown in a text file on the text viewer gedit.

Altering additional settings

Tab *Controller* also allows to setup other parameters related to the simulation. In field *Sample time* (item 5), the sampling time in miliseconds can be configured. The remaining fields *Error file* (item 6), *Reference data file* (item 7), *Sensor file* (item 8) amd *Actuator data file* (item 9) determine the names of the text files where the values of the tracking error, desired path, sensor data and control signal will be logged, stored. These files can be loaded directly to MATLAB. They will available in the directory

```
$HOME/catkin_ws/srcProVANT-Simulator/source/Structure/Matlab
```

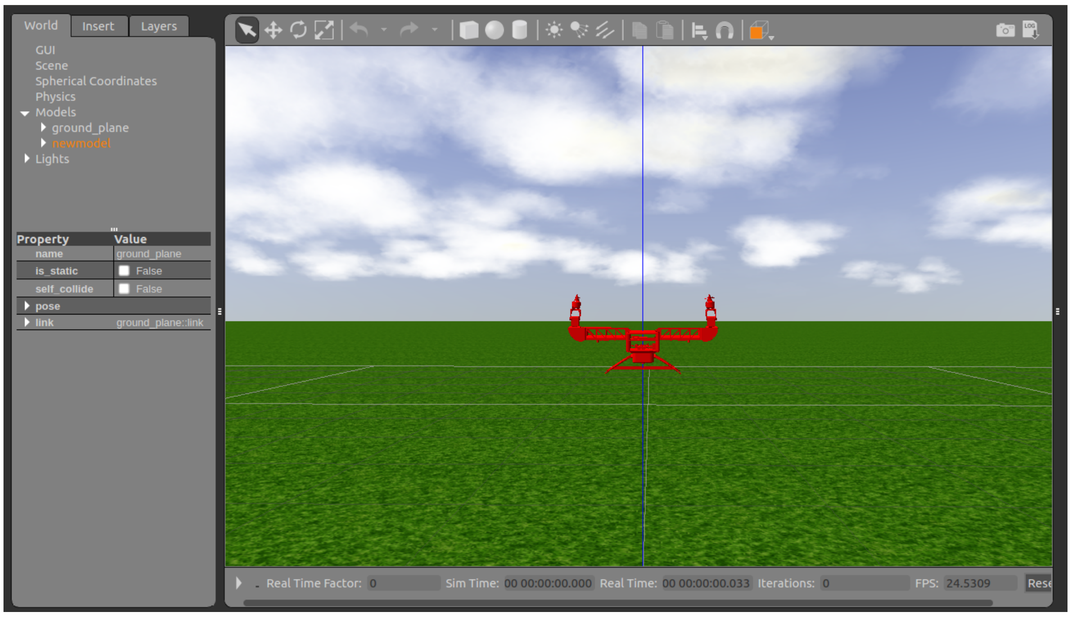
3.5.1 Selecting available instrumentation

Tabs *Sensors* and *Actuators* depicted in Figure 3.8 list, respectively, the names of the sensors and actuators topics to which the controller will have access during simulation **in the presented order**. Those topics are configured during plugin configuration, and will be described in section A.1.2.

3.5.2 Starting the simulation

After setting everything up, the user must press the button *OK* (item 10). The simulation settings window (Figure 3.3) will be shown again. The simulation can then be initialized with the selected UAV model, scenario, control straregy and instrumentation, by pressing the button *Startup Gazebo* (item 4) of Figure 3.3. The simulator Gazebo will be started, as shown in Figure 3.9.

To start the simulation, the user must press the button *Step*. **Button *Play* must not be used**.



Property	Value
name	ground_plane
is_static	False
self_collide	False
pose	
link	ground_plane::link

Real Time Factor: 0 Sim Time: 00 00:00:00.000 Real Time: 00 00:00:00.033 Iterations: 0 FPS: 24.5309

Figure 3.9: Gazebo's home window

Chapter 4

Control strategy design

This chapter describes the procedures required for implementing control straregies in the simulation environment. First, the organization and structure of the files rquired to implement the control strategy are described. Then, the creation process of a new control strategy is presented, so as to illustrate the procedure shown in Section 3.5.

4.1 Organization

The general structure of the control design files is organized as depicted in Figure 4.1:

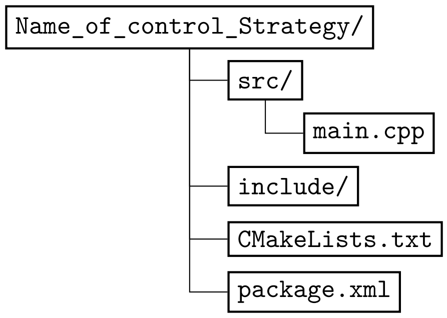


Figure 4.1: Organization of the control design's directory. The folders are represented by the boxes ending in the character `/`, and the files are the names containing extensions.

- File `main.cpp` is where the control stategy's logic is implemented.
- Folder `include/` stores user customized libraries that are included in `main.cpp`'s preamble.
- File `CMakeLists.txt` provides the compiler with information about the directory of the libraries included in the simulator's source codes. Details on how to include new libraries can be found in Appendix C.
- File `package.xml` is needed to store data such as the author's name, email adress, etc. Details on its configuration are presented in Appendix D.

4.2 Standard interface for developing control strategies and template `main.cpp`

The creation of a new control strategy is done by inheriting the standard virtual class `IController`. Being a virtual class, its member functions (methods) must be implemented in the daughter class. Code 4.1 shows such methods.

Method `config()` is executed at the beginning of the simulations, and is used to make initial settings in the control strategy. Method `execute()` is called by the simulator at every sampling period. It is **the method that must contain all the control strategy's logic**. This function's order and amount of input and ouput signals is determined in the interface as described in subsection 3.5.1. Functions `state()`, `error()` and `reference()` are methods that return the values of the error, reference and sensor signals, to be stored in text files. The data stored in `.txt` files can be used to plot graphics of the simulation results using MATLAB, for example.

```
#ifndef ICONTROLLER_HPP
#define ICONTROLLER_HPP

#include "simulator_msgs/SensorArray.h"

class Icontroller
{
        public:
        Icontroller(){};
        virtual ~Icontroller(){};
        virtual void config()=0;
        virtual std::vector<double> execute(simulator_msgs::SensorArray)=0;
        virtual std::vector<double> Reference()=0;
        virtual std::vector<double> Error()=0;
        virtual std::vector<double> State()=0;
};

extern "C" {
        typedef Icontroller* create_t();
        typedef void destroy_t(Icontroller*);
}
#endif
```

Code 4.1: Interface for implementing control strategies

When a new control strategy is created, the simulation environment provides a file `main.cpp` as basic template for the implementation of the control strategy. This file's content is shown in Code 4.2. The next section presents an example of control strategy implementation.

```
#include "Icontroller.hpp"

class demonstracao : public Icontroller
{
        public: demonstracao(){}
        public: ~demonstracao(){}
        public: void config(){}
        public: std::vector<double> execute(simulator_msgs::SensorArray arraymsg)
        {
                std::vector<double> out;
                return out;
        }
        public: std::vector<double> Reference()
        {
                std::vector<double> out;
                return out;
        }
        public: std::vector<double> Error()
        {
                std::vector<double> out;
                return out;
        }
        public: std::vector<double> State()
        {
                std::vector<double> out;
                return out;
        }
};

extern "C"
{
        Icontroller *create(void) {return new demonstracao;}
        void destroy(Icontroller *p) {delete p;}
}
```

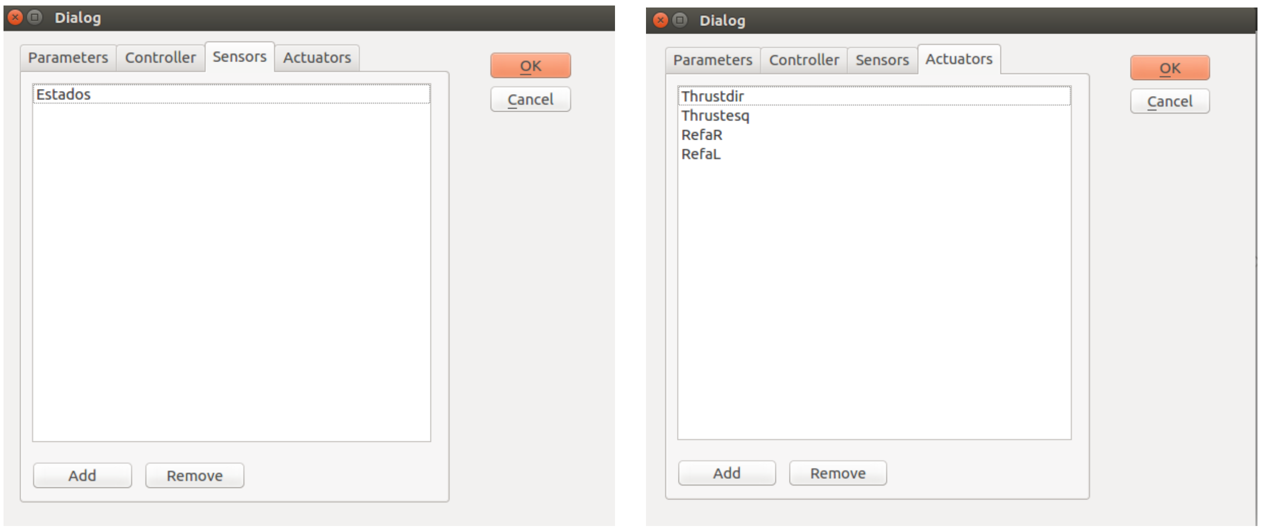
Code 4.2: Control strategy implementation template

4.3 Control strategy implementation example

This section demonstrates the process of implementing a new control strategy. It shows an exemple of using the GUI to create and modigy a new control strategy, and also the compilation process. The example used corresponds to the implementation of a robust control strategy for tracking the load path, carried by a tilt-rotor UAV. More details on the control strategy can be found in Lara et al. (2017).

4.3.1 Configuring the list of available sensors and actuators

In the window described in Section 3.5, when tabs *Sensors* or *Actuators* are selected, one of the windows depicted in Figure 4.2 will show up. In both of them, the user can add and remove instruments from the lists by using buttons *Add* and *Remove*. Furthermore, by double-clicking a name in the list, it is possible to edit the instruments properties. Those lists are of fundamental importance for the control strategy's implementation. The instruments and their respective order will define the input and output data for the controller.



Estados

Thrustdir
Thrustesq
RefaR
RefaL

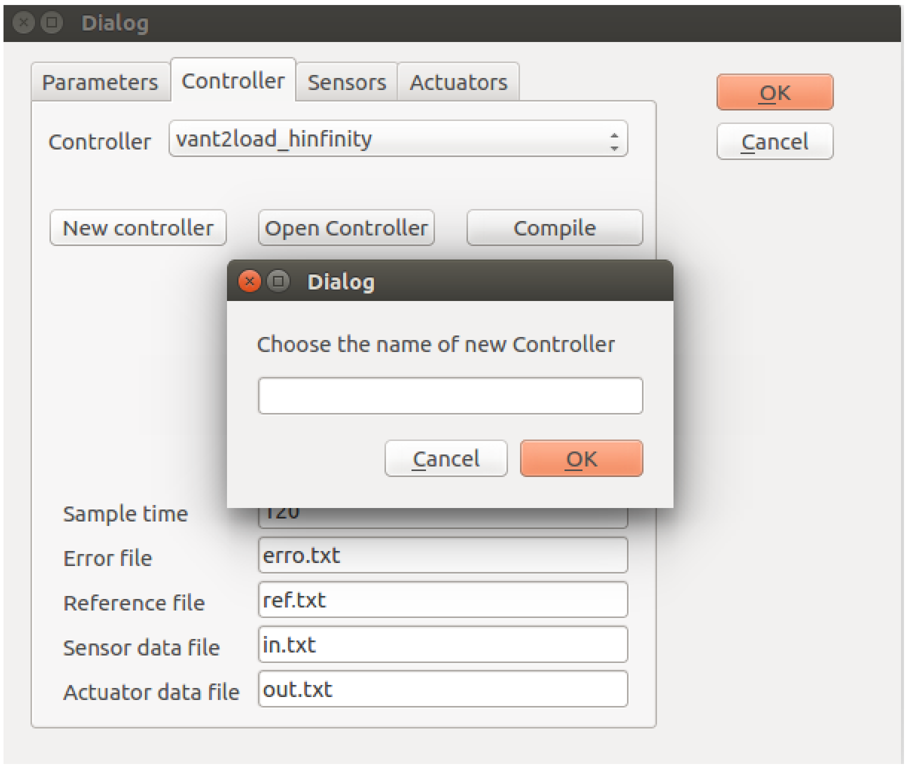
Figure 4.2: Tabs *Sensors* and *Actuators*

In this example, the controller wil receive an array with scalar data containig information provided by a single sensor, whose communication topic is named `Estados`. The controller must return a floating point vector of size 4 with input control signals for the actuators, whose communication topics are in the following order:

1. `Thrustdir`
2. `Thrustesq`
3. `RefaR`
4. `RefaL`

4.3.2 Ceating a new control strategy

To crate a new control strategy, press *New Controller* in tab *Controller*. A new window wil show up, requesting the user for the new controller's name. In this example, the control strategy will be called `vant2load_hinfinity` (Figure 4.3). After confirming the controller's name, a new Nautilus window will be opened with the directory for the created project (Figure 4.4).



Controller: vant2load_hinfinity

Choose the name of new Controller

Sample time: 120
Error file: erro.txt
Reference file: ref.txt
Sensor data file: in.txt
Actuator data file: out.txt

Figure 4.3: Ceating a new control strategy

4.3.3 Implementing the new control strategy

Code 4.3 shows an example of control strategy code implemented in file `main.cpp`. It can be noted in the example that it is necessary to include three libraries:

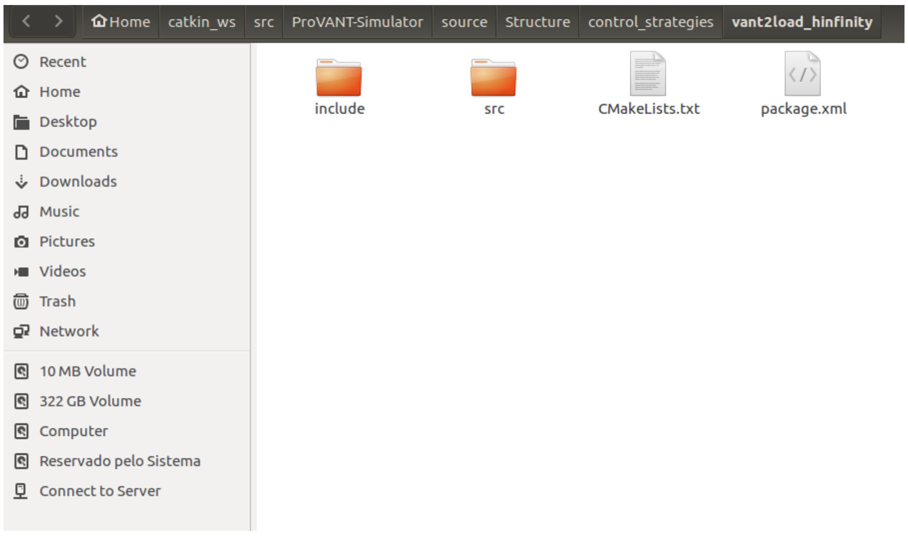


Figure 4.4: Files and directories associated with the new control strategy

```
#include "Icontroller.hpp"
#include <Eigen/Eigen>
#include "simulator_msgs/Sensor.h"

class hinfinity : public Icontroller
{
        private: Eigen::VectorXd Xref; // vetor de referência
        private: Eigen::VectorXd Erro; // vetor de erros
        private: Eigen::VectorXd Input; // sinais de controle
        private: Eigen::MatrixXd K; // matriz de ganhos do controlador
        private: Eigen::VectorXd X; // vetor de estados
        private: double T; // Período de amostragem

        public: hinfinity(): Xref(24), K(4,24), X(24), Erro(24), Input(4)
        {
                T = 0.012;
        }
        public: ~hinfinity(){        }

        public: void config()
        {
                K<< [...] Dados da matriz [...];
        }
        public: std::vector<double> execute(simulator_msgs::SensorArray arraymsg)
        {
                static float count = 0;
                static float xint, x_ant = 0;
                static float yint, y_ant = 0;
                static float zint, z_ant = 0;
                static float yawint, yaw_ant = 0;
                // selecionando dados
                int i = 0;
                simulator_msgs::Sensor msgstates;
                msgstates = arraymsg.values.at(0);
```

```
// Referência
float trajectoryRadius = 2;
float trajectoryHeight = 4*trajectoryRadius;
float trajTime = 80;
float pi = 3.14;
float x = trajectoryRadius*cos((count*T)*2*pi/trajTime);
float xdot = -trajectoryRadius*(2*pi/trajTime)*sin((count*T)*2*pi/trajTime);
float xddot = -trajectoryRadius*(2*pi/trajTime)*(2*pi/trajTime)
                *cos((count*T)*2*pi/trajTime);
float y = trajectoryRadius*sin((count*T)*2*pi/trajTime);
float ydot = trajectoryRadius*(2*pi/trajTime)*cos((count*T)*2*pi/trajTime);
float yddot = -trajectoryRadius*(2*pi/trajTime)*(2*pi/trajTime)
                *sin((count*T)*2*pi/trajTime);
float z = trajectoryHeight+1 - trajectoryHeight*cos((count*T)*2*pi/trajTime);
float zdot = trajectoryHeight*(2*pi/trajTime)*sin((count*T)*2*pi/trajTime);
float zddot = trajectoryHeight*(2*pi/trajTime)*(2*pi/trajTime)
                *cos((count*T)*2*pi/trajTime);
Xref << x,y,z,0,0,0,0.00002965,0.004885,0.004893,0.00484,xdot,ydot,zdot,
        0,0,0,0,0,0,0,0,0,0,0;

//Convertendo velocidade angular
std::vector<double> etadot = pqr2EtaDot(msgstates.values.at(13),
msgstates.values.at(14),
msgstates.values.at(15),
msgstates.values.at(3),
msgstates.values.at(4),
msgstates.values.at(5));

// Integrador Trapezoidal
float x_atual = msgstates.values.at(0) - Xref(0);
xint = xint + (T/2)*(x_atual + x_ant);
x_ant = x_atual;
float y_atual = msgstates.values.at(1) - Xref(1);
yint = yint + (T/2)*(y_atual + y_ant);
y_ant = y_atual;
float z_atual = msgstates.values.at(2) - Xref(2);
zint = zint + (T/2)*(z_atual + z_ant);
z_ant = z_atual;
float yaw_atual = msgstates.values.at(5) - Xref(5);
yawint = yawint + (T/2)*(yaw_atual + yaw_ant);
yaw_ant = yaw_atual;

// vetor de estados aumentado
X << msgstates.values.at(0),//x
msgstates.values.at(1),//y
msgstates.values.at(2),//z
msgstates.values.at(3),//roll
msgstates.values.at(4),//pitch
msgstates.values.at(5),//yaw
msgstates.values.at(8),//g1 x
msgstates.values.at(9),//g2 y
msgstates.values.at(6),//aR
msgstates.values.at(7),//aL
msgstates.values.at(10),//vx
msgstates.values.at(11),//vy
msgstates.values.at(12),//vz
```

```
                etadot.at(0),//droll
                etadot.at(1),//pitch
                etadot.at(2),//yaw
                msgstates.values.at(18),
                msgstates.values.at(19),
                msgstates.values.at(16),
                msgstates.values.at(17),
                xint,
                yint,
                zint,
                yawint;


                // lei de controle
                Erro = X-Xref;
                Input = -K*Erro;

                Eigen::MatrixXd qref(10,1);
                qref << x,y,z,0,0,0,0.00002965,0.004885,0.004893,0.00484;
                Eigen::MatrixXd qrefdot(10,1);
                qrefdot << xdot,ydot,zdot,0,0,0,0,0,0,0;
                Eigen::MatrixXd qrefddot(10,1);
                qrefddot << xddot,yddot,zddot,0,0,0,0,0,0,0;
                Eigen::MatrixXd  varfeedforward  = feedforward::compute(qref,qrefdot,qrefddot);

                // Feedforward
                Input(0) = Input(0) + 12.6005;
                Input(1) = Input(1) + 12.609;
                count++;

                std::vector<float> out2(Input.data(), Input.data() + Input.rows()
                                        * Input.cols());

                std::vector<double> out(out2.size());
                for(int i=0; i<out2.size();i++) out.at(i) = out2.at(i);
                return out;
        }

        public: std::vector<double> Reference()
        {
                std::vector<double> out(Xref.data(), Xref.data() + Xref.rows()
                                        * Xref.cols());
                return out;
        }

        public: std::vector<double> Error()
        {
                std::vector<double> out(Erro.data(), Erro.data() + Erro.rows()
                                        * Erro.cols());
                return out;
        }

        public: std::vector<double> State()
        {
                std::vector<double> out(X.data(), X.data() + X.rows() * X.cols());
                return out;
        }
        private: std::vector<double> pqr2EtaDot(double in_a, double in_b, double in_c,
                                                 double phi, double theta, double psii)
        {
                std::vector<double> out;
                out.push_back(in_a + in_c*cos(phi)*tan(theta) + in_b*sin(phi)*tan(theta));
                out.push_back(in_b*cos(phi) - in_c*sin(phi));
                out.push_back((in_c*cos(phi))/cos(theta) + (in_b*sin(phi))/cos(theta));
                return out;
        }
};

extern "C"
{
        Icontroller *create(void) {
                return new hinfinity;
        }
        void destroy(Icontroller *p) {
                delete p;
        }
}
```

Code 4.3: Example code

- `#include "Icontroller.hpp"` informs to the code the standard interface for creating controllers in the simulation environment;
- `#include <Eigen/Eigen>` imports the functionalities provided by library `Eigen`[1]. This library provides funcitons for making linear algebra operations;
- `#include "simulator_msgs/Sensor.h"` informs the class responsible for the abstraction of the communication pattern between the controller and the sensors.

Attributes `Xref`, `Erro`, `Input`, `K` and `X` are `Eigen` library's vector and matrix structures, responsible for the linear algebra operation, thus allowing for the control strategy's execution. Atrtibute `T` determines the controller's sampling period.

Constructor `hinfinity()` inicializes the class' attibutes, while destructor `~hinfinity()` has no functionality in the simulator's context.

In this example, method `config()` is used to attribute values to the gain matrix `K`, respecting the `Eigen` library's sintax. However, the user can use this space to make any other initial configuration. Finally, the control strategy's logic is implemented in method `execute()`, that is executed at every sampling period, as previously mentioned.

The code begins declaring the (static) variable `xint`, `x_ant`, `yint`, `y_ant`, `zint`, `z_ant`, `yawint` and `yaw_ant`, used to store the values of the integrators impplemented in the contro lstrategy. Next comes the code piece regarding the sensors' data, obtained by vector `arraymsg`. Since only one sensor is available in this example (`Estados`), only the vector's first position is acessed (using the syntax `arraymsg.values.at(0)`). If more than one sensor were to be available, the nth sensor's data would be accessed using `arraymsg.values.at(n-1)` and the order of this vector's elements would be defined in advance in tab *Sensor*, as shown in 3.8. Next, the reference for the controller is defined and the integrators are implemented, using the trapezoidal integration method, for the realization of the control law. Finally, the feedfoward control action is calculated.

Methods `Reference()`, `Error()` and `State()` are used to store the reference, error and state signals, respectively, in the simulator's output text files. Function `pqr2EtaDot()` corresponds to a mapping of part of the information received by the sensors, required for this example's control strategy implementation.

Like function `pqr2EtaDot()`, any additional function required for the control strategy's implementation, which can also be related to filtering algorithms, can be written in `main.cpp`, or in auxiliary files created inside the folder `include/`, shown in 4.4 (in that case, they must also be included in `main.cpp`'s header).

[1]https://eigen.tuxfamily.org

Finally, the following code piece corresponds to the functions required to make sure the control strategy will be loaded in the simulator's execution time. Function `create()` creates an instance of the class that encapsulates the controller, and function `destroy()` is used to destruct the class's instance.

```
extern "C"
{
        Icontroller *create(void) {
                return new hinfinity;
        }
        void destroy(Icontroller *p) {
                delete p;
        }
}
```

4.3.4 Compiling the control strategy's code

After implementing the control strategy, the corresponding code can be compiled by pressing button *Compile* (as explained in Chapter 3, see Figure 3.5 item 4).

Appendix A

Models

The files associated with the UAV models used in ProVANT Simulator are located in the following path:

```
$HOME/catkin_ws/src/provant_simulator/source/Database/simulation_elements/models/real
```

Each model in the simulation environment has a directory with its respective name. This directory contains files that describe the dyanmic, visual, collision and sensorial models, and the control law used. Thus, in case it's necessary to add a new model, or edit an existing one, it must have the UAV's configuration/description files organized as depicted in Figure A.1.

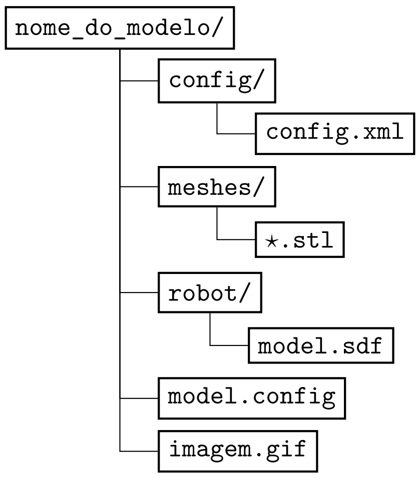


Figure A.1: Directory's organization with UAV model description files

where:

1. File `config.xml` stores the information regarding the controller to be used in the model.
2. File `model.config` describes thte UAV's dynamic, collision and visual models for simulator Gazebo.
3. File `model.config` describes the model's metadata.
4. File `imagem.gif` contains the image used by the garphical interface to illustrate the UAV model (see Figure 3.3, item 2).
5. The directory `meshes` stores the files exported from the CAD tool (used for the UAV's mechanical design), like SolidWorks.

The file `model.config` tells Gazebo where is the file with the model's strucutal data and provides information regarding the models's author, version and description. Figure A.1 illustrates an example of this file. The tags used in `model.config` are:

- <name> The model's name </name>
- <version> The model's version </version>
- <sdf> Path to the file that describes the Gazebo model's dynamic, colision and visual characteristics </sdf>
- <author>
  - <name> The author's name </name>
  - <email> Author's email adress </email>

  </author>
- <description> Brief description of the model </description>

```
<?xml version="1.0"?>
<model>
    <name>vant</name>
    <version>1.0</version>
    <sdf version='1.5'>robot/model.sdf</sdf>
    <author>
        <name>provant</name>
        <email>provant@ufmg.br</email>
    </author>
    <description>
        The UAV version 3.0 of the provant project
    </description>
</model>
```

Code A.1: Example content for file `model.config`

The file `config.xml` stores all the settings regarding the control structure to be used during the simulation, such as sensors, actuatores, control laws and sampling period. ith the purpose of helping the user to edit this file, the graphical interface provides tools for this task, so that the user won't have to directly access this file. Code A.2 illustrates an example of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <TopicoStep>Step</TopicoStep>
    <Sampletime>12</Sampletime>
    <Strategy>libvant3_lqr.so</Strategy>
    <RefPath>ref.txt</RefPath>
    <Outputfile>out.txt</Outputfile>
    <InputPath>in.txt</InputPath>
    <ErroPath>erro.txt</ErroPath>
    <Sensors>
        <Device>Estados</Device>
    </Sensors>
    <Actuators>
        <Device>ThrustR</Device>
        <Device>ThrustL</Device>
        <Device>TauR</Device>
        <Device>TauL</Device>
        <Device>Elevdef</Device>
        <Device>Ruddef</Device>
    </Actuators>
</config>
```

Code A.2: Example of `config.xml` file

The tags used in file `config.xml` are:

- `<TopicoStep>` The sincronization topic between the simulator and the controller. **This tag's value must not be changed.** `</TopicoStep>`
- `<Sampletime>` The controller's smapling period, in milliseconds `</Sampletime>`
- `<Strategy>` Name of the library associated to the implementation of the control strategy to be used along with this model `</Strategy>`
- `<RefPath>` Name of the file where the setpoint signal data is stored `</RefPath>`
- `<Outputfile>` Name of the file where the control signal data is stored `</Outputfile>`
- `<InputPath>` Name of the file where the sensor data is stored `</InputPath>`
- `<ErroPath>` Name of the file where the error vector data is stored `</ErroPath>`
- `<Sensors>` Name of the sensor topics to which the control strategy will have access (in the specified order) `</Sensors>`
- `<Actuators>` Name of the actuator topics to which the control strategy will have access (the controller must return a vector with the same amount of data and in the order specified here) `</Actuators>`

File `model.sdf` provides to Gazebo the UAV model's information in XML format following the SDF format (Figure A.3 illustrates an example of this file). The next section describes more thoroughly the basic structure of an SDF file.

A.1 The SDF file

Before presenting the basic configuration of an SDF file, it's necessary to introduce a few concepts.

In the simulator, a model correspond to a mechanical system, which can be formed my one or multiple rigid bodies[1]. Thus, as in a manipulator, the bodies in the simulator are called links. Child links are connected to parent links by joints. Child links are rigid bodies whose movements are restricted by the connection (joint) with bodies called parent links. The links have inertial, visual and collision properties.

As for joints, they impose restrictions to the relative movement between two links and have properties such ase joint type (prismatic, revolute, etc.), movement limits (position ans speed), friction, etc. Code A.3 illustrates an example of an SDF file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.4">
    <model name="modelo">
        <link name="corpo">
            ...
        </link>
        <link name="servo">
            ...
        </link>
        <joint name="corpo_servo">
            ...
        </joint>
    </model>
</sdf>
```

Code A.3: Description of a model in file `model.sdf`

The tags used are:

- `<link>` Describes a link, specifying its name `</link>`
- `<joint>` Describes a joint, specifying its name `</joint>`

Each link in the model has trhee types of description for the simulator: the kinematic, visual and collision descriptions. The configuration structure of a link in an SDF file has athe format depicted in code A.4:

[1]By assuming a body as rigid, the elasticity and deformation effects are neglected

```
<link name="servodir">
    <pose>0.02E-3 -277.61E-3 56.21E-3 -0.0872665 0 0</pose>
    <inertial>
        ...
    </inertial>
    <collision name="servodircollision"> <!--opcional-->
        ...
    </collision>
    <visual name="servodirvisual"> <!--opcional-->
        ...
    </visual>
</link>
```

Code A.4: Description of a link in file `model.sdf`

The tags used are:

- `<pose>` Link's pose `</pose>`
- `<inertial>` Link's inertial properties `</inertial>`
- `<collision>` Link's collision properties. The collision models for the UAVs used in the simulation environment are obtained from CAD files. `</collision>`
- `<visual>` Visual characteristics such as color and shape. The visual models for the UAVs used in the simulation environment are obtained from CAD files, except for the color, which is specified separately. `</visual>`

Links inertial parameters: The user must inform each link's inertial parameters inside the tag `inertial`. The required information are the mass, the center of mass' relative position and the inertia tensor. In Code A.5 an exemple configuration of a link's inertial parameters in format SDF is illustrated.

```
<inertial>
    <mass>0.0809439719362664</mass>
    <pose>
        -3.60859273452335E-10 -0.000226380714807978 0.0594780519701684 0 0 0
    </pose>
    <inertia>
        <ixx>3.88267747087835E-06</ixx>
        <ixy>6.03219085082653E-06</ixy>
        <ixz>-2.78471406661236E-12</ixz>
        <iyy>0.000104858690365283</iyy>
        <iyz>7.0486590219062E-07</iyz>
        <izz>8.31755564684115E-05</izz>
    </inertia>
</inertial>
```

Code A.5: Description of inertial characteristics in file `model.sdf`

The tags used are:

- `<mass>` The link's mass `</mass>`
- `<pose>` The center of mass' position relative to its main coordinate system `</pose>`
- `<inertia>` The link's inertia tensor `</inertia>`

Link's collision properties: In order for collision effects to be applied to the link, the user must describe the link's shape in file `model.sdf`. There are many ways to describe it, but this guide presents only the method used in the UAV models in ProVANT Simulator, which consists of importing files created with CAD tools, such as SolidWorks. Code A.6 shows an example descriptioin of a link's visual parameters using an STL file:

```
        <collision name="servodircollision">
            <pose>0 0 0 0 0 0</pose>
            <geometry>
                <mesh>
                    <uri>model://vant_2comcarga/meshes/servodir.STL</uri>
                </mesh>
            </geometry>
        </collision>
```

Code A.6: Description of collision characteristics in file `model.sdf`

The tags used are:

- `<pose>` The collision model's pose relative to the link's coordinates' origin `</pose>`
- `<geometry>`
    - `<mesh>`
        - `<uri>` Path to the mesh file, relative to the model's directory, obtained by exporting from SolidWorks `</uri>`

      `</mesh>`

  `</geometry>`

Links visual properties: In order for the link o be visualized during the simulation, the user must describe the link's visual parameters in file `model.sdf`. Just like in the previous case, there are several description methods, but this guide illustrates only the one used in the simulation environment's UAVs, which consists of importing files created from CAD tools. Code A.7 Shows an example description of a link's visual parameters using an STL file:

```
        <visual name="servodirvisual">
            <pose>0 0 0 0 0 0</pose>
            <geometry>
                <mesh>
                    <uri>model://vant_2comcarga/meshes/servodir.STL</uri>
                </mesh>
            </geometry>
            <material>
                <ambient>0 0 0 0</ambient>
                <diffuse>1 1 1 1</diffuse>
                <specular>0.1 0.1 0.1 1</specular>
                <emissive>0 0 0 0</emissive>
            </material>
        </visual>
```

Code A.7: Description of visual characteristics in file `model.sdf`

The tags used are:

- `<pose>` The visual model's pose relative to the link's coordinates' origin `</pose>`
- `<geometry>`
    - `<mesh>`
        - `<uri>` Path to the mesh file, relative to the model's directory, obtained by exporting from SolidWorks `</uri>`

      `</mesh>`

  `</geometry>`
- `<material>`
    - `<ambient>` Ambient color `</ambient>`

- <diffuse> Diffuse color </diffuse>
- <specular> Specular color </specular>
- <emissive> Emissive color </emissive>

</material>

A.1.1 Joint description

Ther joint types in the simulator are:

- **revolute**: Rotation movement;
- **gearbox**: Revolute joint with transmission gears between links with different torque and speed ratios;
- **revolute2**: Joint composed by two revolute joints in series;
- **prismatic**: Prismatic joint;
- **universal**: Joint behaving as an articulated sphere;
- **piston**: Joint behaving as a combination of a revolute and a prismatic joint.

An example of a joint's configuration structure is shown in Code A.8.

```
<joint name="aR" type="revolute">
    <pose>0 0 0 0 0 0</pose>
    <parent>corpo</parent>
    <child>servodir</child>
    <axis>
        <xyz>0 0.9962 -0.0872</xyz>
        <limit>
            <lower>-1.5</lower>
            <upper>1.5</upper>
            <effort>2</effort>
            <velocity>0.5</velocity>
        </limit>
        <dynamics>
            <damping>0</damping>
            <friction>0</friction>
        </dynamics>
    </axis>
</joint>
```

Code A.8: Joint description in file `model.sdf`

The tags used are:

- <pose> Child link's pose relative to the parent link </pose>
- <parent> Parent link's name </parent>
- <axis> Unit vector corresponding to the joint's rotation axis (expressed in the model's coordinate system if the SDF version is 1.4, or in the child link's coordinate system if the version is 1.6) </axis>
- <lower> Lower limit to the joint's position (in radians if it's a revolute joint or meters if it's prismatic) </lower>
- <upper> Upper limit to the joint's position (in radians if it's a revolute joint or meters if it's prismatic) </upper>
- <velocity> Joint's speed limit (in rad/s if it's a revolute joint or m/s if it's prismatic) </velocity>
- <effort> Joint's effort limit (in N·m if it's a revolute joint or N if it's prismatic) </effort>
- <damping> Vicous friction coefficient </damping>
- <friction> Static friction coefficient </friction>

A.1.2 Plugin description

Plugins are dynamic libraries loaded during the simulator's initialization, using the configurations stored in the model description file (SDF file). These libraries are used to implement the sensors and actuators in the simulation environment.

ProVANT Simulator only uses two of Gazebo's plugin types for the UAV model's control and monitoring: Model and Sensor.

Model plugins: Model plugins are dynamic libraries which control and monitor the UAV model's simulation variables. With them it's possible to create customized sensors and actuators. To insert a model plugin in file `model.sdf`, the user must add `<plugin>` tags defining its name, the dynamic library's name and the internal tags required to configure the plugin. Code A.9 exemplifies the insertion process.

The options for model plugins available in ProVANT Simulator are detailed in appendix B.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.4">
    <model name="modelo">
        <link name="corpo">
            ...
        </link>
        <link name="servo">
            ...
        </link>
        <joint name="corpo_servo">
            ...
        </joint>
        <plugin name="plugin_servo">
            ...
        </plugin>
    </model>
</sdf>
```

Code A.9: Insertion of model plugins in file `model.sdf`

Sensor plugins: Sensor plugins are dynamic lybraries to simulate sensors, used by ProVANT Simulator to measure data from a UAV model. To insert sensor plugins, the user must add `<sensor>` tags defining its name and the internal tags required to configure the plugin. Code A.10 exemplifies the insertion process.

The sensor available in ProVANT Simulator are GPS, IMU, sonar and magnetometer (more details on their configuration in file `model.sdf`). However, those sensors don't have a communication interface with ROS, since they broadcast their date via Gazebo topics. To broadcast these data to ROS topics the user must add, along with the sensor plugins, model plugins. Such plugins are specified in appendix B.

Note: In order for the sensor to work properly, the user must adjust the sampling rate to exactly the inverse of the simulation step (e.g. 1000 Hz).

A.1.3 Communication messages' standard

As a standard, all plugins and sensor available in the simulation environment make their provide their data using the same data structure. This data structure is abstracted in ROS through messages. Messages are simple data structures containing typed fields. The sensor plugins' standard message is illustrated below, where:

- `header`: provides the time instant in which the data as obtained, the frame and the sequential ID
- `name`: provides the name of the instrument which provided the message
- `values`: vector containing the data provided by the sensors

```
        <link name="servodir">
            <pose>0.02E-3 -277.61E-3 56.21E-3 -0.0872665 0 0</pose>
            <inertial>
                ...
            </inertial>
            <collision name="servodircollision"><!--opcional-->
                ...
            </collision>
            <visual name="servodirvisual"> <!--opcional-->
                ...
            </visual>
            <sensor name="servosensor"> <!--opcional-->
                ...
            </sensor>
            <sensor name="servosensor2"> <!--opcional-->
                ...
            </sensor>
        </link>
```

Code A.10: Insertion of sensor plugins in file `model.sdf`

```
- Header header
- string name
- float64[] values
```

The controller, in turn, receives a type of message which stores the messages of all sensors from a given simulation step in the same place and in a user-defined order, as described in section 3.5.1. This type is illustrated below:

```
- Header header
- string name
- float64[] values
```

Appendix B

ProVANT Simulator's existing plugins

B.1 Model plugins

This section shows the configuration of the model plugins which don't work with data exchange between Gazebo and ROS. The configuration information is shown in Tables B.1, B.2, B.3, B.4, B.5, B.6, B.7 and B.8.

Table B.1:
Brushless motor plugin configuration

Description:	Simulates lift forces due to the two blades' rotation driven by a tilt-rotor's brushless motors	
File:	`libgazebo_ros_brushless_plugin.so`	
Configuration:	`<topic_FR>`	Topic with the value of the right blade's lift force
	`<topic_FL>`	Topic with the value of the left blade's lift force
	`<LinkDir>`	Link corresponding to the right blade
	`<LinkEsq>`	Link corresponding to the left blade

Table B.2:
Servomotor plugin configuration

Description:	Simulates servomotors with torque or position operation modes	
File:	`libgazebo_servo_motor_plugin.so`	
Configuration:	`<NameOfJoint>`	Joint to be controlled by the servomotor
	`<TopicSubscriber>`	Topic with the reference values for the servomotor
	`<TopicPublisher>`	Topic with the servo's sensor data (position and speed)
	`<Modo>`	Servomotor's operation mode (options: `Torque` or `Posição`)

Table B.3:
State space plugin configuration

Description:	Senses the tilt-rotor UAV's state vector $(x, y, z, \phi, \theta, \psi, \alpha_R, \alpha_L, \dot{x}, ty, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \dot{\alpha}_R, \dot{\alpha}_L)$	
File:	`libgazebo_AllData_plugin.so`	
Configuration:	`<NameOfTopic>`	Topic from which the user can obtain the information
	`<NameOfJointR>`	Right servomotor's joint
	`<NameOfJointL>`	Left servomotor's joint
	`<bodyname>`	Link corresponding to the servomotor's main body

Table B.4:
State space load plugin configuration

Description:	Senses the state vector for a tilt-rotor UAV with load transportation $(x, y, z, \phi, \theta, \psi, \alpha_R, \alpha_L, \lambda_x, \lambda_y, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \dot{\alpha}_R, \dot{\alpha}_L, \dot{\lambda}_x, \dot{\lambda}_y)$	
File:	`libgazebo_AllData2_plugin.so`	
Configuration:	`<NameOfTopic>`	Topic from which the user can obtain the information
	`<NameOfJointR>`	Right servomotor's joint
	`<NameOfJointL>`	Left servomotor's joint
	`<NameOfJoint_X>`	Joint corresponding to the load's degree of freedom around the X axis
	`<NameOfJoint_Y>`	Joint corresponding to the load's degree of freedom around the Y axis
	`<bodyname>`	Link corresponding to the servomotor's main body

Table B.5:
Temperature plugin configuration

Description:	Senses temperature and air pressure with noise	
File:	`libgazebo_ros_temperature.so`	
Configuration:	`<Topic>`	Topic from which the user can obtain the information
	`<TempOffset>`	Error offset for noisy temperature data
	`<TempStandardDeviation>`	Error standard deviation for noisy temperature data
	`<BaroOffset>`	Error offset for noisy pressure data
	`<BaroStandardDeviation>`	Error standard deviation for noisy pressure data
	`<maxtemp>`	Maximum temperature value
	`<mintemp>`	Minimum temperature value
	`<maxbaro>`	Maximum pressure value
	`<minbaro>`	Minimum pressure value
	`<Nbits>`	Number of bits used in the digitalization

Table B.6:
UniversalJointSensor plugin configuration

Description:	Senses all the data Gazebo provides for a joint (angle, angular velocity and torque)	
File:	`libgazebo_ros_universaljoint.so`	
Configuration:	`<NameOfTopic>`	Topic from which the user can obtain the information
	`<NameOfJoint>`	Joint to be sensed
	`<Axis>`	Joint's (first) rotation axis
	`<Axis2>`*	Joint's second rotation axis (*used only for joints with two degrees of freedom)

Table B.7:
UniversalLinkSensor plugin configuration

Description:	Senses all data Gazebo provides for a link	
File:	`libgazebo_ros_universallink.so`	
Configuration:	`<NameOfTopic>`	Topic from which the user can obtain the information
	`<NameOfLink>`	Link to be sensed

Table B.8:
Order in which data is presented in plugin UniversalLinkSensor

0	Relative pose in X
1	Relative pose in Y
2	Relative pose in Z
3	Relative pose in ϕ
4	Relative pose in θ
5	Relative pose in ψ
6	Relative speed in X
7	Relative speed in Y
8	Relative speed in Z
9	Relative linear acceleration in X
10	Relative linear acceleration in Y
11	Relative linear acceleration in Z
12	Relative force in X
13	Relative force in Y
14	Relative force in Z
15	Relative angular speed in X
16	Relative angular speed in Y
17	Relative angular speed in Z
18	Relative angular acceleration in X
19	Relative angular acceleration in Y
20	Relative angular acceleration in Z
21	Relative mechanical torque in X
22	Relative mechanical torque in Y
23	Relative mechanical torque in Z
24	Global pose in X
25	Global pose in Y
26	Global pose in Z
27	Global pose in ϕ
28	Global pose in θ
29	Global pose in ψ
30	Global speed in X
31	Global speed in Y
32	Global speed in Z
33	Global linear acceleration in X
34	Global linear acceleration in Y
35	Global linear acceleration in Z
36	Global force in X
37	Global force in Y
38	Global force in Z
39	Global angular speed in X
40	Global angular speed in Y
41	Global angular speed in Z
42	Global angular acceleration in X
43	Global angular acceleration in Y
44	Global angular acceleration in Z
45	Global mechanical torque in X
46	Global mechanical torque in Y
47	Global mechanical torque in Z
48	Center of gravity's global linear speed in X
49	Center of gravity's global linear speed in Y
50	Center of gravity's global linear speed in Z
51	Center of gravity's global linear pose in X
52	Center of gravity's global linear pose in Y
53	Center of gravity's global linear pose in Z

B.2 Model plugins used along sensor plugins

This section shows the configuration of the model plugins which work with data exchange between Gazebo and ROS. The configuration information is shown in Tables B.9, B.10, B.11 and B.12.

Table B.9:
Model plugin for transmitting GPS data from Gazebo topics to ROS topics

Description:	Transmits GPS sensor data to ROS	
File:	`libgazebo_ros_gps.so`	
Configuration:	`<gazebotopic>`	Gazebo topic where the GPS publishes its data
	`<rostopic>`	ROS topic to be read by the controller
	`<link>`	Link to which the GPS is attached

Table B.10:
Model plugin for transmitting IMU data from Gazebo topics to ROS topics

Description:	Transmits IMU data to ROS	
File:	`libgazebo_ros_imu.so`	
Configuration:	`<gazebotopic>`	Gazebo topic where the IMU publishes its data
	`<rostopic>`	ROS topic to be read by the controller
	`<link>`	Link to which the IMU is attached

Table B.11:
Model plugin for transmitting sonar data from Gazebo topics to ROS topics

Description:	Transmits sonar data to ROS	
File:	`libgazebo_ros_sonar.so`	
Configuration:	`<gazebotopic>`	Gazebo topic where the sonar publishes its data
	`<rostopic>`	ROS topic to be read by the controller
	`<link>`	Link to which the sonar is attached

Table B.12:
Model plugin for transmitting magnetometer data from Gazebo topics to ROS topics

Description:	Transmits magnetometer data to ROS	
File:	`libgazebo_ros_magnetometro.so`	
Configuration:	`<gazebotopic>`	Gazebo topic where the magnetometer publishes its data
	`<rostopic>`	ROS topic to be read by the controller
	`<link>`	Link to which the magnetometer is attached

Appendix C

CMakeLists.txt

This section was extracted from http://wiki.ros.org/catkin/CMakeLists.txt on August 08, 2017. Visit it for further information.

C.1 Overview and structure of file `CMakeLists.txt`

The file CMakeLists.txt is the input to the CMake build system for building software packages. This file **must follow the following format and order**.

1. Required CMake Version (`cmake_minimum_required`)
2. Package Name (`project()`)
3. Find other CMake/Catkin packages needed for build (`find_package()`)
4. Enable Python module support (`catkin_python_setup()`)
5. Message/Service/Action Generators (`add_message_files(), add_service_files(), add_action_files()`)
6. Invoke message/service/action generation (`generate_messages()`)
7. Specify package build info export (`catkin_package()`)
8. Libraries/Executables to build (`add_library()/add_executable()/target_link_libraries()`)
9. Tests to build (`catkin_add_gtest()`)
10. Install rules (`install()`)

C.2 CMake Version

Every catkin CMakeLists.txt file must start with the required version of CMake needed. Catkin requires version 2.8.3 or higher.

```
cmake_minimum_required(VERSION 2.8.3)
```

C.3 Package name

The next item is the name of the ROS package. In the following example, the package is called *robot_brain*.

```
project(robot_brain)
```

Note: In CMake you can reference the project name anywhere later in the CMake script by using the variable `${PROJECT_NAME}`.

C.4 Finding dependent CMake packages

We need to then specify which other CMake packages that need to be found to build our project using the CMake `find_package` function. There is always at least one dependency on catkin:

```
find_package(catkin REQUIRED)
```

If your project depends on other wet packages, they are automatically turned into components (in terms of CMake) of catkin. Instead of using `find_package` on those packages, if you specify them as components, it will make life easier. For example, if you use the package `nodelet`.

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

Note: You should only `find_package` components for which you want build flags. You should not add runtime dependencies.

C.4.1 Command `find_package()`

If a package is found by CMake through `find_package`, it results in the creation of several CMake environment variables that give information about the found package. These environment variables can be utilized later in the CMake script. The environment variables describe where the packages exported header files are, where source files are, what libraries the package depends on, and the paths of those libraries. The names always follow the convention of `<PACKAGE NAME>_<PROPERTY>`:

`<NAME>_FOUND`: Set to true if the library is found, otherwise false
`<NAME>_INCLUDE_DIRS` or `<NAME>_INCLUDES`: The include paths exported by the package
`<NAME>_LIBRARIES` or `<NAME>_LIBS`: The libraries exported by the package
`<NAME>_DEFINITIONS`: Definitions exported by the package

C.4.2 Why are catkin packages specified as components?

Catkin packages are not really components of catkin. Rather the components feature of CMake was utilized in the design of catkin to save you significant typing time.

For catkin packages, if you `find_package` them as components of catkin, this is advantageous as a single set of environment variables is created with the `catkin_` prefix. For example, let us say you were using the package `nodelet` in your code. The recommended way of finding the package is:

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

This means that the include paths, libraries, etc exported by nodelet are also appended to the `catkin_` variables. For example, `catkin_INCLUDE_DIRS` contains the include paths not only for catkin but also for `nodelet` as well! This will come in handy later.

We could alternatively `find_package nodelet` on its own:

```
find_package(nodelet)
```

This means the `nodelet` paths, libraries and so on would not be added to `catkin_` variables.

This results in `nodelet_INCLUDE_DIRS`, `nodelet_LIBRARIES`, and so on. The same variables are also created using

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

C.4.3 Boost

If using C++ and Boost, you need to invoke `find_package()` on Boost and specify which aspects of Boost you are using as components. Boost is a set of libraries for C++ which offers support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions and unit testing. For example, if you wanted to use Boost threads, you would say:

```
find_package(Boost REQUIRED COMPONENTS thread)
```

C.5 `catkin_package()`

Command `catkin_package()` specifies catkin-specific information for the compiler.

This function must be called before declaring any targets with `add_library()` or `add_executable()`. The function has 5 optional arguments:

`INCLUDE_DIRS`:	The exported include paths for the package
`LIBRARIES`:	The exported libraries from the project
`CATKIN_DEPENDS`:	Other catkin projects that this project depends on
`DEPENDS`:	Non-catkin projects that this project depends on
`CFG\_EXTRAS`:	Additional configuration options

As an example:

```
catkin_package(INCLUDE_DIRS include
    LIBRARIES ${PROJECT_NAME}
    CATKIN_DEPENDS roscpp nodelet
    DEPENDS eigen opencv)
```

This indicates that the folder `include/` within the package folder is where exported headers go. The CMake environment variable `${PROJECT_NAME}` evaluates to whatever you passed to the `project()` function earlier, in this case it will be `robot_brain`. `roscpp` and `nodelet` are packages that need to be present to build/run this package, and `eigen` and `opencv` are system dependencies that need to be present to build/run this package.

C.6 Specifying build targets

Build targets can take many forms, but usually they represent one of two possibilties:

Executable Target:	programs we can run
Library Target:	libraries that can be used by executable targets at build and/or runtime

C.6.1 Target naming

It is very important to note that the names of build targets in catkin must be unique regardless of the folders they are built/installed to. This is a requirement of CMake. However, unique names of targets are only necessary internally to CMake. One can have a target renamed to something else using the `set_target_properties()` function.

Example:

```
set_target_properties(rviz_image_view
                      PROPERTIES OUTPUT_NAME image_view
                      PREFIX "")
```

This will change the name of the target `rviz_image_view` to `image_view` in the build and install outputs.

C.6.2 Custom output directory

While the default output directory for executables and libraries is usual set to a reasonable value it must be customized in certain cases, i.e. a library containing Python bindings must be placed in a different folder to be importable in Python.

Example:

```
set_target_properties(python_module_library
PROPERTIES LIBRARY_OUTPUT_DIRECTORY ${CATKIN_DEVEL_PREFIX}${CATKIN_PACKAGE_PYTHON_DESTINATION})
```

C.6.3 Include paths and library paths

Prior to specifying targets, you need to specify where resources can be found for said targets, specifically header files and libraries

```
Include paths
Library paths
include_directories(<dir1>, <dir2>, ..., <dirN>)
link\_directories(<dir1>, <dir2>, ..., <dirN>)
```

a) `include_directories()`

The argument to `include_directories` should be the `*_INCLUDE_DIRS` variables generated by your `find_package` calls and any additional directories that need to be included. If you are using catkin and Boost, your `include_directories()` call should look like:

```
include_directories(include ${Boost_INCLUDE_DIRS} ${catkin_INCLUDE_DIRS})
```

The first argument `include` indicates that the `include/` directory within the package is also part of the path.

b) `link_directories()`

The CMake `link_directories()` function can be used to add additional library paths, however, this is not recommended. All catkin and CMake packages automatically have their link information added when they are `find_packaged`. Simply link against the libraries in `target_link_libraries()`.

Example:

```
link_directories(~/my_libs)
```

C.6.4 Executable targets

To specify an executable target that must be built, we must use the `add_executable()` CMake function.

```
add_executable(myProgram src/main.cpp src/some_file.cpp src/another_file.cpp)
```

This will build a target executable called `myProgram` which is built from 3 source files: `src/main.cpp`, `src/some_file.cpp` and `src/another_file.cpp`.

C.6.5 Library targets

The `add_library()` CMake function is used to specify libraries to build. By default catkin builds shared libraries.

```
add_library(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
```

C.6.6 `target_link_libraries`

Use the `target_link_libraries()` function to specify which libraries an executable target links against. This is done typically after an `add_executable()` call. Add `${catkin_LIBRARIES}` if ros is not found.

Syntax:

```
target_link_libraries(<executableTargetName>, <lib1>, <lib2>, ... <libN>)
```

Example:

```
add_executable(foo src/foo.cpp)
add_library(moo src/moo.cpp)
target_link_libraries(foo moo)  -- This links foo against libmoo.so
```

Note that there is no need to use `ink_directories()`in most use cases as that information is automatically pulled in via `find_package()`.

C.7 Messages, services, and action targetss

Message (`.msg`), service (`.srv`) and action (`.action`) files in ROS require a special preprocessor build step before being built and used by ROS packages. The point of these macros is to generate programming language-specific files so that one can utilize messages, services, and actions in their programming language of choice. The build system will generate bindings using all available generators (e.g. gencpp, genpy, genlisp, etc).

There are three macros provided to handle messages, services, and actions respectively:

```
add_message_files
add_service_files
add_action_files
```

These macros must then be followed by a call to the macro that invokes generation:

```
generate_messages()
```

Important prerequisites/constraints

These macros must come textbfbefore the `catkin_package()` macro in order for generation to work correctly.

```
find_package(catkin REQUIRED COMPONENTS ...)
add_message_files(...)
add_service_files(...)
add_action_files(...)
generate_messages(...)
catkin_package(...)
...
```

Your `catkin_package()` macro must have a `CATKIN_DEPENDS` dependency on `message_runtime`.

```
catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)
```

You must use `find_package()` for the package `message_generation`, either alone or as a component of catkin:

```
find_package(catkin REQUIRED COMPONENTS message_generation)
```

Your `package.xml` file must contain a build dependency on `message_generation` and a runtime dependency on `message_runtime`. This is not necessary if the dependencies are pulled in transitively from other packages.

If you have a target which (even transitively) depends on some other target that needs messages/services/actions to be built, you need to add an explicit dependency on target `catkin_EXPORTED_TARGETS`, so that they are built in the correct order. This case applies almost always, unless your package really doesn't use any part of ROS. Unfortunately, this dependency cannot be automatically propagated. (`some_target` is the name of the target set by `add_executable()`):

```
add_dependencies(some_target ${catkin_EXPORTED_TARGETS})
```

If you have a package which builds messages and/or services as well as executables that use these, you need to create an explicit dependency on the automatically-generated message target so that they are built in the correct order. (`some_target` is the name of the target set by `add_executable()`):

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS})
```

If you your package satisfies both of the above conditions, you need to add both dependencies, i.e.:

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
```

C.7.1 Example

If your package has two messages in a directory called `msg` named `MyMessage1.msg` and `MyMessage2.msg` and these messages depend on `std_msgs` and `sensor_msgs`, a service in a directory called `srv` named `MyService.srv`, defines executable `message_program` that uses these messages and service, and an executable called `does_not_use_local_mes` which uses some parts of ROS, but not the messages/service defined in this package, then you will need the following in your `CMakeLists.txt`:

```
# Get the information about this package's buildtime dependencies
find_package(catkin REQUIRED
    COMPONENTS message_generation std_msgs sensor_msgs)

# Declare the message files to be built
add_message_files(FILES
    MyMessage1.msg
    MyMessage2.msg
)

# Declare the service files to be built
add_service_files(FILES
    MyService.srv
)

# Actually generate the language-specific message and service files
generate_messages(DEPENDENCIES std_msgs sensor_msgs)

# Declare that this catkin package's runtime dependencies
catkin_package(
 CATKIN_DEPENDS message_runtime std_msgs sensor_msgs
)

# define executable using MyMessage1 etc.
add_executable(message_program src/main.cpp)
add_dependencies(message_program ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

# define executable not using any messages/services provided by this package
add_executable(does_not_use_local_messages_program src/main.cpp)
add_dependencies(does_not_use_local_messages_program ${catkin_EXPORTED_TARGETS})
```

If, additionally, you want to build actionlib actions, and have an action specification file called `MyAction.action` in the `action` directory, you must add `actionlib_msgs` to the list of components which are `find_packaged` with catkin and add the following call before the call to `generate_messages(...)`:

```
add_action_files(FILES
    MyAction.action
)
```

Furthermore the package must have a build dependency on `actionlib_msgs`.

C.8 Enabling Python module support

If your ROS package provides some Python modules, you should create a `setup.py` file and call

```
catkin_python_setup()
```

before the call to `generate_messages()` and `catkin_package()`.

C.9 Unit tests

There is a catkin-specific macro for handling gtest-based unit tests called `catkin_add_gtest()`.

```
catkin_add_gtest(myUnitTest test/utest.cpp)
```

C.10 Optional step: specifying installable targets

After build time, targets are placed into the devel space of the catkin workspace. However, often we want to install targets to the system so that they can be used by others or to a local folder to test a system-level installation. In other words, if you want to be able to do a "make install" of your code, you need to specify where targets should end up.

This is done using the CMake `install()` function which takes as arguments:

`TARGETS`: Which targets to install
`ARCHIVE`DESTINATION : Static libraries and DLL (Windows) .lib stubs
`LIBRARY`DESTINATION : Non-DLL shared libraries and modules
`RUNTIME`DESTINATION : Executable targets and DLL (Windows) style shared libraries

Take as an example:

```
install(TARGETS ${PROJECT_NAME}
    ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
    LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
    RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Besides these standard destination some files must be installed to special folders, i.e. a library containing Python bindings must be installed to a different folder to be importable in Python:

```
install(TARGETS python_module_library
    ARCHIVE DESTINATION ${CATKIN_PACKAGE_PYTHON_DESTINATION}
    LIBRARY DESTINATION ${CATKIN_PACKAGE_PYTHON_DESTINATION}
)
```

C.10.1 Installing Python Executable Scripts

For Python code, the install rule looks different as there is no use of the `add_library()` and `add_executable()` functions so as for CMake to determine which files are targets and what type of targets they are. Instead, use the following in your `CMakeLists.txt` file:

```
catkin_install_python(PROGRAMS scripts/myscript
    DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

Detailed information about installing python scripts and modules, as well as best practices for folder layout can be found in the catkin manual.

If you only install Python scripts and do not provide any modules, you need neither to create the above mentioned `setup.py` file, nor to call `catkin_python_setup()`.

C.10.2 Installing header files

Header files must also be installed to the `include` folder, This is often done by installing the files of an entire folder (optionally filtered by filename patterns and excluding SVN subfolders). This can be done with an install rule that looks as follows:

```
install(DIRECTORY include/${PROJECT_NAME}/
    DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
    PATTERN ".svn" EXCLUDE
)
```

or if the subfolder under include does not match the package name:

```
install(DIRECTORY include/
    DESTINATION ${CATKIN_GLOBAL_INCLUDE_DESTINATION}
    PATTERN ".svn" EXCLUDE
)
```

Installing `roslaunch` files or other resources

Other resources like launchfiles can be installed to `${CATKIN_PACKAGE_SHARE_DESTINATION}`:

```
install(DIRECTORY launch/
    DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}/launch
    PATTERN ".svn" EXCLUDE)
```

Appendix D

package.xml

This section was extracted from http://wiki.ros.org/catkin/package.xml on August 25,2017. Visit it for further information.

D.1 Overviewl

The package manifest is an XML file called `package.xml` that must be included with any catkin-compliant package's root folder. This file defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages.

Your system package dependencies are declared in `package.xml`. If they are missing or incorrect, you may be able to build from source and run tests on your own machine, but your package will not work correctly when released to the ROS community. Others depend on this information to install the software they need for using your package.

D.2 Format 2 (recommended)

This is the recommended format for new packages. It is also recommended that older format 1 packages be migrated to format 2. For instructions on migrating from format 1 to format 2, see Migrating from Format 1 to Format 2 in the catkin API docs.

The full documentation for format 2 can be found in the catkin API docs.

D.2.1 Basic structure

Each `package.xml` file has the `<package>` tag as the root tag in the document.

```
<package format="2">

</package>
```

D.2.2 Required tags

There is a minimal set of tags that need to be nested within the `<package>` tag to make the package manifest complete.

`<name>`: The name of the package
`<name>`: The version number of the package (required to be 3 dot-separated integers)
`<name>`: A description of the package contents
`<name>`: The name of the person(s) that is/are maintaining the package
`<name>`: The software license(s) (e.g. GPL, BSD, ASL) under which the code is released

As an example, here is package manifest for a fictional package called `foo_core`.

```
<package format="2">
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@osrf.org">Ivana Bildbotz</maintainer>
        <license>BSD</license>
</package>
```

D.2.3 Dependencies

The package manifest with minimal tags does not specify any dependencies on other packages. Packages can have six types of dependencies:

- **Build Dependencies** specify which packages are needed to build this package. This is the case when any file from these packages is required at build time. This can be including headers from these packages at compilation time, linking against libraries from these packages or requiring any other resource at build time (especially when these packages are `find_package`d in CMake). In a cross-compilation scenario build dependencies are for the targeted architecture.
- **Build Export Dependencies** specify which packages are needed to build libraries against this package. This is the case when you transitively include their headers in public headers in this package (especially when these packages are declared as (`CATKIN_`)`DEPENDS` in `catkin_package()` in CMake).
- **Execution Dependencies** specify which packages are needed to run code in this package. This is the case when you depend on shared libraries in this package (especially when these packages are declared as (`CATKIN_`)`DEPENDS` in `catkin_package()` in CMake).
- **Test Dependencies** specify only additional dependencies for unit tests. They should never duplicate any dependencies already mentioned as build or run dependencies.
- **Build Tool Dependencies** specify build system tools which this package needs to build itself. Typically the only build tool needed is catkin. In a cross-compilation scenario build tool dependencies are for the architecture on which the compilation is performed.
- **Documentation Tool Dependencies** specify documentation tools which this package needs to generate documentation.

These six types of dependencies are specified using the following respective tags:

`<depend>` specifies that a dependency is a build, export, and execution dependency. This is the most commonly used dependency tag.
`<buildtool_depend>`
`<build_depend>`
`<build_export_depend>`
`<exec_depend>`
`<test_depend>`
`<doc_depend>`

All packages have at least one dependency, a build tool dependency on catkin as the following example shows.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@osrf.org">Ivana Bildbotz</maintainer>
        <license>BSD</license>
        <buildtool_depend>catkin</buildtool_depend>
</package>
```

A more realistic example that specifies build, exec, test, and doc dependencies could look as follows.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
        <license>BSD</license>
        <url>http://ros.org/wiki/foo_core</url>
        <author>Ivana Bildbotz</author>
        <buildtool_depend>catkin</buildtool_depend>
        <depend>roscpp</depend>
        <depend>std_msgs</depend>
        <build_depend>message_generation</build_depend>
        <exec_depend>message_runtime</exec_depend>
        <exec_depend>rospy</exec_depend>
        <test_depend>python-mock</test_depend>
        <doc_depend>doxygen</doc_depend>
</package>
```

D.2.4 Metapackages

It is often convenient to group multiple packages as a single logical package. This can be accomplished through metapackages. A metapackage is a normal package with the following export tag in the `package.xml`:

```
<export>
        <metapackage />
</export>
```

Other than a required `<buildtool_depends>` dependency on catkin, metapackages can only have execution dependencies on packages of which they group.

Additionally a metapackage has a required, boilerplate `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 2.8.3)
project(<PACKAGE_NAME>)
find_package(catkin REQUIRED)
catkin_metapackage()
```

Note: replace `<PACKAGE_NAME>` with the name of the metapackage.

D.2.5 Additional tags

`<url>`: A URL for information on the package, typically a wiki page on ros.org.
`<author>`: The author(s) of the package

D.3 Format 1 (legacy))

Older catkin pakages use format 1. If the `<package>` tag has no `format` attribute, it is a format 1 package. Use format 2 for new packages.

The format of `package.xml` is straightforward.

D.3.1 Basic structure

Each `package.xml` file has the `<package>` tag as the root tag in the document.

```
<package>

</package>
```

D.3.2 Required tags

There are a minimal set of tags that need to be nested within the `<package>` tag to make the package manifest complete.

`<name>`: The name of the package
`<name>`: The version number of the package (required to be 3 dot-separated integers)
`<name>`: A description of the package contents
`<name>`: The name of the person(s) that is/are maintaining the package
`<name>`: The software license(s) (e.g. GPL, BSD, ASL) under which the code is released

As an example, here is package manifest for a fictional package called `foo_core`.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
        <license>BSD</license>
</package>
```

D.3.3 Build, run, and test dependencies

O manifesto do pacote com tags mínimas não especifica nenhuma dependência em outros pacotes. Pacotes podem ter quatro tipos de dependências:

- **Build Dependencies** specify build system tools which this package needs to build itself. Typically the only build tool needed is catkin. In a cross-compilation scenario build tool dependencies are for the architecture on which the compilation is performed.
- **Build Export Dependencies** specify which packages are needed to build this package. This is the case when any file from these packages is required at build time. This can be including headers from these packages at compilation time, linking against libraries from these packages or requiring any other resource at build time (especially when these packages are `find_packaged` in CMake). In a cross-compilation scenario build dependencies are for the targeted architecture.
- **Execution Dependencies** specify which packages are needed to run code in this package, or build libraries against this package. This is the case when you depend on shared libraries or transitively include their headers in public headers in this package (especially when these packages are declared as `(CATKIN_)DEPENDS` in `catkin_package()` in CMake).
- **Test Dependencies** specify only additional dependencies for unit tests. They should never duplicate any dependencies already mentioned as build or run dependencies.

These four types of dependencies are specified using the following respective tags:

- `<buildtool_depend>`
- `<build_depend>`
- `<run_depend>`
- `<test_depend>`

All packages have at least one dependency, a build tool dependency on catkin as the following example shows.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
        <license>BSD</license>
        <buildtool_depend>catkin</buildtool_depend>
</package>
```

A more realistic example that specifies build, runtime, and test dependencies could look as follows.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
        <license>BSD</license>
        <url>http://ros.org/wiki/foo_core</url>
        <author>Ivana Bildbotz</author>
        <buildtool_depend>catkin</buildtool_depend>
        <build_depend>message_generation</build_depend>
        <build_depend>roscpp</build_depend>
        <build_depend>std_msgs</build_depend>
        <run_depend>message_runtime</run_depend>
        <run_depend>roscpp</run_depend>
        <run_depend>rospy</run_depend>
        <run_depend>std_msgs</run_depend>
        <test_depend>python-mock</test_depend>
</package>
```

D.3.4 Metapackages

It is often convenient to group multiple packages as a single logical package. This can be accomplished through metapackages. A metapackage is a normal package with the following export tag in the `package.xml`:

```
<export>
        <metapackage />
</export>
```

Other than a required `<buildtool_depend>` dependency on catkin, metapackages can only have run dependencies on packages of which they group.

Additionally a metapackage has a required, boilerplate `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 2.8.3)
project(<PACKAGE_NAME>)
find_package(catkin REQUIRED)
catkin_metapackage()
```

Note: replace `<PACKAGE_NAME>` with the name of the metapackage.

D.3.5 Additional tags

`<url>`: A URL for information on the package, typically a wiki page on ros.org.
`<author>`: The author(s) of the package

Bibliography

Alfaro, R. (2016). Predictive Control Strategies for Unmanned Aerial Vehicles in Cargo Transportation Tasks. Master's thesis, Universidade federal de Santa Catarina.

Cardoso, D. N. (2016). Adaptative Control Strategies For Improved Forward Flight of a Tilt-Rotor UAV. Master's thesis, Universidade Federal de Minas Gerais.

de Almeida Neto, M. M. (2014). Control Strategies of a Tilt-rotor UAV for Load Transportation. Master's thesis, Universidade Federal de Minas Gerais.

Donadel, R. (2015). Modeling and Control of a Tiltrotor Unmanned Aerial Vehicle for Path Tracking. Master's thesis, Universidade Federal de Santa Catarina.

Lara, A. V., S.Rego, B., Raffo, G. V., & Arias-Garcia, J. (2017). Desenvolvimento de um ambiente de simulação de VANTs Tilt-rotor para testes de estratégias de controle. *Anais do XIII SBAI*.

Rego, B. S. (2016). Path Tracking Control of a Suspended Load Using a Tilt-Rotor UAV. Master's thesis, Universidade Federal de Minas Gerais.